UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Andraž Drčar

Prevajanje javanskih programov z vstavljeno zložno kodo

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Javanska zložna koda je zbirni jezik za javanski navidezni stroj. Poleg javanskih programov se v javansko zložno kodo lahko prevajajo tudi drugi jeziki, na primer, Python, Ruby, Groovy, Scala, C in drugi. Poznavanje javanske zložne kode pomaga razumeti način delovanja javanskega navideznega stroja in lahko pomembno vpliva na pisanje optimalne programske kode.

V diplomskem delu preučite delovanje javanskega navideznega stroja in javanske zložne kode. Napišite razširitev javanskega prevajalnika, ki bo omogočal vstavljanje zložne kode neposredno v javanske programe (jasm bloki). Vaš program naj znotraj jasm blokov podpira vse ukaze zložne kode ter omogoča dostop do vseh spremenljivk, ki so zapisane v naboru konstant.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Andraž Drčar, z vpisno številko **63020032**, sem avtor diplomskega dela z naslovom:

Prevajanje javanskih programov z vstavljeno zložno kodo

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 10. septembra 2014 Podpis avtorja:

Na tem mestu bi se rad zahvalil vsem, ki bi mi utegnili zameriti, če se jim na tem mestu ne bi zahvalil!

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
RISC	Reduced Instruction Set Computing	računalnik s skrčenim naborom ukazov
JDK	Java Development Kit	razvojni komplet za Javo
JRE	Java Runtime Environment	javansko izvajalno okolje
JVM	Java Virtual Machine	javanski navidezni stroj

Povzetek

Java je eden izmed bolj razširjenih programskih jezikov. Poznan je predvsem zaradi svoje sistemske neodvisnosti, ki je dosežena s pomočjo javanskega virtualnega stroja razvitega za specifične sisteme. Ti stroji sledijo natančnim navodilom, kako izvajati zložno kodo iz prevedenih datotek. Na drugi strani pa natančnih navodil za prevajanje v zložno kodo ni, zato programer nima vpliva na prevedeno kodo.

V diplomskem delu je opisan razvoj razširjenega prevajalnika za Javo. Prevajalnik poleg standardnih ukazov sprejme tudi bloke, ki vsebujejo ukaze zložne kode. V prvem delu je kratka predstavitev delovanja javanskega navideznega stroja in sestava datotek z zložno kodo. V nadaljevanju so opisane zahteve razširjenega prevajalnika, čemur sledi opis rešitve, ki je v grobem razdeljen na pet korakov. Pri vsakem koraku so podrobno opisane osnovne ideje, njihove morebitne pomanjkljivosti, težave med razvojem in končne rešitve. Zaključek je sestavljen iz analize končnega prevajalnika z opisi nekaterih možnosti za razširitev in nadgradnjo.

Ključne besede: Java, zložna koda, prevajalnik

Abstract

Java is one of the top programming languages known for its platform independency, which is reached by using platform specific Java Virtual Machines (JVM). Each JVM follows strict rules how class files containing the bytecode are parsed and executed. However, there are no such rules for the compilation part and the programmer has no influence on the compiled code.

The thesis describes the development of the extended compiler for Java. In addition to the standard commands this compiler also supports usage of blocks that contain Java bytecode. The first part is a brief presentation of the Java Virtual Machine and the composition of translated files. The following describes the requirements of the extended compiler followed by description of the solution, which is roughly divided into five steps. Each step contains the basic ideas, their possible shortcomings, problems during development and the presentation of final solution. The conclusion is drawn from the analysis of the final compiler and descriptions of some options to expand and upgrade the product.

Keywords: Java, bytecode, compiler

Poglavje 1 Uvod

Java je eden izmed bolj razširjenih programskih jezikov, ki je znan predvsem po svoji sistemski neodvisnosti. To pomeni, da prevedene kode po selitvi na drug sistem ni treba popravljati ali ponovno prevajati. Za to poskrbi javansko okolje s svojim navideznim strojem (*angl. Java Virtual Machine - JVM*), ki je razvit za vrsto različnih procesorjev in operacijskih sistemov [1]. Vsi ti stroji sledijo natančnim pravilom, kako brati in izvajati prevedeno kodo, ki je zapisana kot zaporedje ukazov zložne kode. Medtem ko je izvajanje natančno predpisano, za prevajanje ne obstajajo standardizirana pravila. Ker obstaja veliko število različnih sistemov, obstaja tudi veliko prevajalnikov različnih razvijalcev. Obstaja torej možnost, da se isti program prevede v različno zaporedje ukazov zložne kode. Posledica tega je, da programer nima nikakršnega vpliva na prevedeno datoteko in nima možnosti za optimizacijo. Medtem ko prevajalniki za nekatere ostale programske jezike omogočajo uporabo procesorskih ukazov, programiranje z zložno kodo v Javi ni podprto. Tabela 1 prikazuje uporabo procesorskih ukazov v programskem jeziku C.

```
#include <stdio.h>

int main() {
    /* Sešteje 10 in 20 in rezultat shrani v register %eax */
    __asm__ ( "movl $10, %eax;"
              "movl $20, %ebx;"
              "addl %ebx, %eax;"
    );
    return 0 ;
}
```

Tabela 1 - Primer uporabe procesorskih ukazov v programskem jeziku C.

Ob pregledu spleta smo ugotovili, da se uporabniki po forumih zanimajo za tovrstne rešitve, a za zdaj obstajajo samo orodja, ki znajo analizirati in predstaviti sestavo zložne kode. Takšni orodji sta na primer vtičnika za eclipse Bytecode Outline plugin for Eclipse [2] ter Bytecode Visualizer [7]. Zato smo prišli na idejo, razviti prevajalnik za Javo, ki bo poleg običajnih ukazov omogočal tudi uporabo blokov z zložno kodo. Za razvoj takšnega prevajalnika je dobro razumeti nekatere osnove delovanja JVM, predvsem pa se je bilo treba podrobno seznaniti s sestavo in ukazi zložne kode, kar je opisano v naslednjem poglavju. Za tem je

opisano, kaj naj bi prevajalnik omogočal ter kakšne omejitve smo sprejeli pri izdelavi prevajalnika. Sledi opis razvoja prevajalnika, kjer so opisane osnovne ideje, težave in njihove rešitve pri posameznih sklopih prevajalnika. Na koncu sledi še analiza izdelanega prevajalnika z opisi nekaterih možnosti za razširitev.

Poglavje 2 Delovanje JVM in zložna koda

Za lažje razumevanje opisa razvoja razširjenega prevajalnika je dobro, da se najprej seznanimo z osnovami delovanja javanskega navideznega stroja ter sestavo datotek z zložno kodo kot tudi samimi ukazi zložne kode.

2.1 Delovanje JVM

JVM je neke vrste navidezni procesor s svojim naborom ukazov. Sprva je bil razvit za izvajanje prevedene kode programskega jezika Java, kasneje pa so se pojavili tudi prevajalniki, ki tvorijo zložno kodo za JVM iz drugih programskih jezikov. Skupaj z javanskim programskim vmesnikom (*angl. Application Programming Interface - API*) JVM tvori javansko izvajalno okolje (*angl. Java Runtime Environment - JRE*). JVM je sestavljen iz enote za preverjanje zložne kode, enote za delo in upravljanje s pomnilnikom ter tolmača. Enota za preverjanje zložne kode najprej preveri, da vsi skoki znotraj programa kažejo na veljavno lokacijo ter da so vse spremenljivke ustrezno inicializirane. Prav tako skrbi tudi za dostope do zaščitenih delov kode. Kot pomnilnik JVM uporablja kopico, kamor se shranjujejo vsi objekti - primerki razredov. Za čiščenje tega dela pomnilnika skrbi čistilec pomnilnika (*angl. garbage collector*). V preostalem delu pomnilnika se nahajajo tabele konstant, kode metod ter po en sklad za vsako izvajajočo se nit. Zadnja komponenta je tolmač, ki ukaze zložne kode prevede in posreduje dejanskemu procesorju sistema. Od leta 2014 se v JVM za doseganje večje hitrosti izvajanja po večini uporablja dinamično prevajanje [6]. To pomeni, da se zložna koda prevaja v ukaze za ustrezno platformo med samim izvajanjem, kar omogoča večjo hitrost delovanja. Toda vse opisano velja za točno določen JVM, saj struktura in podrobnosti delovanja JVM niso natančno določena. Podobno kot to velja tudi za prevajalnike, ki tvorijo prevedene datoteke z zložno kodo, je opis JVM mnogo bolj abstrakten. Kot pravijo pri razvijalcu, z natančnejšo definicijo JVM niso želeli omejevati ustvarjalnosti različnih razvijalcev in so jim pri realizaciji pustili proste roke [3]. JVM je tako definiran kot stroj, ki je zmožen brati prevedene datoteke z zložno kodo in izvajati ukaze, ki so v njih zapisani. Nabor ukazov ter njihova funkcionalnost in zgradba prevedenih datotek sta edini dve stvari, ki sta natančno določeni. Oglejmo si torej še ti dve komponenti programskega jezika Java.

2.2 Sestava datotek z zložno kodo

Prevedena datoteka z zložno kodo ne vsebuje nobenih podatkov o naslovih in nobenih oznak pozicij, na katere bi se lahko sklicevali med delovanjem programa. Vsebino datoteke lahko preberemo samo, če natančno poznamo njeno sestavo. Vsakič je treba datoteko brati od začetka in s pomočjo štetja in vsebine bajtov izluščiti njeno vsebino.

Velikost	Opis
4B	CAFEBABE - konstanta za datoteko class
2B	podverzija
2B	verzija
2B	število konstant (cpc)
različno	deskriptor konstant (cpc-1 zapisov)
2B	zastavice dostopa
2B	ime trenutnega razreda
2B	ime nadrazreda
2B	število vmesnikov (ic)
različno	deskriptor vmesnikov (ic zapisov)
2B	število polj (fc)
različno	deskriptor polj (fc zapisov)
2B	število metod (mc)
različno	deskriptor metod (mc zapisov)
2B	število atributov (ac)
različno	deskriptor atributov (ac zapisov)

Tabela 2 - Sestava datoteke z zložno kodo.

Tabela 2 prikazuje splošen opis sestave datoteke tipa `class`, sledijo pa še podrobnejši opisi deskriptorjev. Za delovanje prevajalnika sta najbolj pomembna deskriptor konstant in deskriptor metod. Pri drugih je dovolj, da vemo, koliko bajtov moramo preskočiti, ko pridemo do njihovega mesta.

2.2.1 Deskriptor konstant

Deskriptor konstant vsebuje vse vrednosti znakovnih konstant, ki jih uporabljamo v programu. Poleg tega vsebuje tudi številske konstante, ki niso predstavljive s številom bitov, ki so nam na voljo za atribute ukaza. Pod znakovne konstante sodijo tako imena spremenljivk, metod in razredov, kot tudi običajne vrednosti spremenljivk tipa `String`. Med zapisi najdemo še sklic na metode, vmesnike in polja ter zapis o imenu in tipu, ki se uporablja tako za metode kot za polja. Prvi bajt je oznaka, ki nam pove kakšnega tipa je konstanta. Odvisno od tipa tej številki sledi različno število bajtov, ki predstavljajo vrednost konstante. Kakšen je format katerega tipa konstante, je zapisano v tabeli Tabela 3.

Št.	Tip konstante	Velikost in pomen
1	Utf8	2 bajta za dolžino, sledijo znaki vrednosti
3	Integer	4 bajte - predznačeno število v dvojiškem komplementu
4	Float	4 bajte - število v plavajoči vejici IEEE 754
5	Long	8 bajtov - predznačeno število v dvojiškem komplementu
6	Double	8 bajtov - število v plavajoči vejici IEEE 754
7	Class	2 bajta - sklic na ime razreda
8	String	2 bajta - sklic na vrednost znakovne spremenljivke
9	sklic polja	4 bajte - sklic na razred, sklic na deskriptor
10	sklic metode	4 bajte - sklic na razred, sklic na deskriptor
11	sklic vmesnika	4 bajte - sklic na razred, sklic na deskriptor
12	Deskriptor	4 bajte - 2bajta za sklic na ime, 2 bajta za sklic na tip

Tabela 3 - Tabela z opisom zapisov v deskriptorju konstant.

2.2.2 Deskriptor vmesnikov

Najbolj preprost deskriptor je deskriptor vmesnikov. Vsebuje namreč sklice na vmesnike iz tabele konstant, ki jih naš razred implementira ali razširja. Vsak zapis je dolg dva bajta. Koliko zapisov vsebuje, nam pove številka pred deskriptorjem.

2.2.3 Deskriptor polj

Deskriptor polj opisuje polja, uporabljena v našem razredu. Sestavljen je iz tabel, ki opisujejo vsako polje posebej. Zgradbo tabele prikazuje Tabela 4. Več o zgradbi in vsebini atributov bomo povedali kasneje.

Velikost	Pomen
2B	Zastavice dostopa
2B	Sklic na ime
2B	Sklic na deskriptor
2B	Število atributov
različno	Atributi

Tabela 4 - Prikaz sestave tabele v deskriptorju polj.

2.2.4 Deskriptor metod

Deskriptor metod je za nas najbolj pomemben del prevedene datoteke. V njem so namreč ukazi zložne kode, v katere so bili prevedeni javanski ukazi znotraj posameznih metod. Poleg ukazov vsebuje tudi najbolj pomemben atribut. To je tabela lokalnih spremenljivk, kjer so zapisane vse lokalne spremenljivke posamezne metode in njihova vidljivost. Deskriptor metod je, podobno kot deskriptor polj, sestavljen iz tabel. V vsaki tabeli je opis ene metode, zgradba tabele pa je enaka kot pri deskriptorju polj in jo prikazuje Tabela 5.

Velikost	Pomen
2B	Zastavice dostopa
2B	Sklic na ime
2B	Sklic na deskriptor
2B	Število atributov
različno	Atributi

Tabela 5 - Prikaz sestave tabele v deskriptorju metod.

Med atributi v tabeli metode se nahaja atribut `Code`, ki vsebuje vse pomembne podatke. Atribut se kot vsi drugi, začne z indeksom na ime v tabeli konstant, ki zasede dva bajta. Sledijo štirje bajti, ki nam povedo, kako velika je vsebina atributa. Vsebina atributa `Code` je opisana v tabeli Tabela 6.

Velikost	Pomen
2B	Maksimalna potrebna velikost sklada
2B	Število lokalnih spremenljivk
2B	Dolžina kode
različno	Ukazi zložne kode metode
2B	Dolžina tabele izjem
Različno	Vsebina tabele izjem
2B	Število atributov
Različno	Atributi

Tabela 6 - Sestava atributa `Code`.

Drugi pomembni del je tabela lokalnih spremenljivk, ki jo v kodi prevedeni z vsebino za razhroščevanje, najdemo med atributi atributa `Code`. Po indeksu, ki kaže na konstanto `LocalVariableTable`, se zopet nahajajo štirje bajti z velikostjo. Prva dva bajta vsebine predstavljata število tabel. Tabela 7 prikazuje sestavo tabel z opisom lokalnih spremenljivk.

Velikost	Pomen
2B	Začetek veljavnosti (številka bajta v zložni kodi)
2B	Dolžina veljavnosti (v bajtih)
2B	Sklic na ime spremenljivke
2B	Sklic na deskriptor
2B	Indeks lokalne spremenljivke (zaporedna številka)

Tabela 7 - Sestava tabele s podatki o lokalnih spremenljivkah.

2.2.5 Deskriptor atributov

Deskriptor atributov je sestavljen iz opisov atributov. Prva dva bajta sta referenca na ime atributa iz tabele konstant. Sledijo štirje bajti, ki nam povejo dolžino vsebine atributa. Sledi vsebina atributa. Kot smo ugotovili že pri atributoma `Code` in `LocalVariableTable`, imajo vsi atributi enako zgradbo. Vseh možnih atributov ne bomo naštevali, saj za delovanje

razširjenega prevajalnika niso pomembni. Treba je vedeti le, koliko jih je in kolikšna je velikost njihove vsebine, kar izvemo iz dodatnih bajtov med opisi. Vsebino atributov, ki niso pomembni za delovanje razširjenega prevajalnika, lahko preskočimo oziroma jo prepišemo v končno datoteko. Več o atributih in njihovem pomenu je zapisano v dokumentaciji [4].

2.3 Ukazi zložne kode

JVM bi lahko označili kot procesor tipa RISC, saj so ukazi kratki in preprosti. Vsi ukazi so dolgi po en bajt, saj JVM vsebuje le 205 ukazov. Ker JVM uporablja sklad, je večina ukazov brez operandov. Poleg tega so med ukazi zložne kode tudi skrajšani ukazi z najbolj pogostimi konstantami, ki pripomorejo k še krajši kodi prevedenih datotek. Tako lahko namesto ukaza `iload` z atributom `1` uporabimo skrajšani ukaz `iload_1`, ki namesto dveh bajtov zasede le enega. V tabeli Tabela 8 so predstavljeni vsi ukazi zložne kode z operacijskimi kodami, opisom operandov, vplivu na sklad ter opisom delovanja.

Tabela 8 - Seznam ukazov zložne kode

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
nop	00		ni sprememb	ne naredi ničesar
aconst_null	01		→ null	potisne referenco za null na sklad
iconst_m1	02		→ -1	potisne vrednost int -1 na sklad
iconst_0	03		→ 0	potisne vrednost int 0 na sklad
iconst_1	04		→ 1	potisne vrednost int 1 na sklad
iconst_2	05		→ 2	potisne vrednost int 2 na sklad
iconst_3	06		→ 3	potisne vrednost int 3 na sklad
iconst_4	07		→ 4	potisne vrednost int 4 na sklad
iconst_5	08		→ 5	potisne vrednost int 5 na sklad
lconst_0	09		→ 0L	potisne vrednost long 0 na sklad
lconst_1	0A		→ 1L	potisne vrednost long 1 na sklad
fconst_0	0B		→ 0.0f	potisne vrednost float 0.0 na sklad
fconst_1	0C		→ 1.0f	potisne vrednost float 1.0 na sklad
fconst_2	0D		→ 2.0f	potisne vrednost float 2.0 na sklad
dconst_0	0E		→ 0.0	potisne vrednost double 0.0 na sklad
dconst_1	0F		→ 1.0	potisne vrednost double 1.0 na sklad
bipush	10	1: bajt	→ vrednost	potisne bajt na sklad kot vrednost int
sipush	11	2: bajt1, bajt2	→ vrednost	potisne 2 bajta na sklad
ldc	12	1: indeks	→ vrednost	potisne vrednost konstante #indeks (String, int, float) na sklad
ldc_w	13	2: indeks	→ vrednost	potisne vrednost konstante #indeks (String, int, float) na sklad (indeks je dolg 2 bajta)
ldc2_w	14	2: indeks	→ vrednost	potisne vrednost konstante #indeks (double, long) na sklad (indeks je dolg 2 bajta)
iload	15	1: indeks	→ vrednost	naloži vrednost int iz lokalne spremenljivke #indeks

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
lload	16	1: indeks	→ vrednost	naloži vrednost long iz lokalne spremenljivke #indeks
fload	17	1: indeks	→ vrednost	naloži vrednost float iz lokalne spremenljivke #indeks
dload	18	1: indeks	→ vrednost	naloži vrednost double iz lokalne spremenljivke #indeks
aload	19	1: indeks	→ referenca objekta	naloži referenco iz lokalne spremenljivke #indeks
iload_0	1A		→ vrednost	naloži vrednost int iz lokalne spremenljivke 0
iload_1	1B		→ vrednost	naloži vrednost int iz lokalne spremenljivke 1
iload_2	1C		→ vrednost	naloži vrednost int iz lokalne spremenljivke 2
iload_3	1D		→ vrednost	naloži vrednost int iz lokalne spremenljivke 3
lload_0	1E		→ vrednost	naloži vrednost long iz lokalne spremenljivke 0
lload_1	1F		→ vrednost	naloži vrednost long iz lokalne spremenljivke 1
lload_2	20		→ vrednost	naloži vrednost long iz lokalne spremenljivke 2
lload_3	21		→ vrednost	naloži vrednost long iz lokalne spremenljivke 3
fload_0	22		→ vrednost	naloži vrednost float iz lokalne spremenljivke 0
fload_1	23		→ vrednost	naloži vrednost float iz lokalne spremenljivke 1
fload_2	24		→ vrednost	naloži vrednost float iz lokalne spremenljivke 2
fload_3	25		→ vrednost	naloži vrednost float iz lokalne spremenljivke 3
dload_0	26		→ vrednost	naloži vrednost double iz lokalne spremenljivke 0
dload_1	27		→ vrednost	naloži vrednost double iz lokalne spremenljivke 1
dload_2	28		→ vrednost	naloži vrednost double iz lokalne spremenljivke 2
dload_3	29		→ vrednost	naloži vrednost double iz lokalne spremenljivke 3
aload_0	2A		→ referenca objekta	naloži referenco iz lokalne spremenljivke 0
aload_1	2B		→ referenca objekta	naloži referenco iz lokalne spremenljivke 1
aload_2	2C		→ referenca objekta	naloži referenco iz lokalne spremenljivke 2
aload_3	2D		→ referenca objekta	naloži referenco iz lokalne spremenljivke 3
iaload	2E		referenca polja, indeks → vrednost	naloži vrednost int iz polja
laload	2F		referenca polja, indeks → vrednost	naloži vrednost long iz polja
faload	30		referenca polja, indeks → vrednost	naloži vrednost float iz polja
daload	31		referenca polja, indeks → vrednost	naloži vrednost double iz polja
aaload	32		referenca polja, indeks → vrednost	naloži referenco iz polja na sklad
baload	33		referenca polja, indeks → vrednost	naloži vrednost boolean iz polja
caload	34		referenca polja, indeks → vrednost	naloži vrednost char iz polja

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
saload	35		referenca polja, indeks → vrednost	naloži vrednost short iz polja
istore	36	1: indeks	vrednost →	shrani vrednost int v lokalno spremenljivko #indeks
lstore	37	1: indeks	vrednost →	shrani vrednost long v lokalno spremenljivko #indeks
fstore	38	1: indeks	vrednost →	shrani vrednost float v lokalno spremenljivko #indeks
dstore	39	1: indeks	vrednost →	shrani vrednost double v lokalno spremenljivko #indeks
astore	3A	1: indeks	referenca objekta →	shrani referenco v lokalno spremenljivko #indeks
istore_0	3B		vrednost →	shrani vrednost int v lokalno spremenljivko 0
istore_1	3C		vrednost →	shrani vrednost int v lokalno spremenljivko 1
istore_2	3D		vrednost →	shrani vrednost int v lokalno spremenljivko 2
istore_3	3E		vrednost →	shrani vrednost int v lokalno spremenljivko 3
lstore_0	3F		vrednost →	shrani vrednost long v lokalno spremenljivko 0
lstore_1	40		vrednost →	shrani vrednost long v lokalno spremenljivko 1
lstore_2	41		vrednost →	shrani vrednost long v lokalno spremenljivko 2
lstore_3	42		vrednost →	shrani vrednost long v lokalno spremenljivko 3
fstore_0	43		vrednost →	shrani vrednost float v lokalno spremenljivko 0
fstore_1	44		vrednost →	shrani vrednost float v lokalno spremenljivko 1
fstore_2	45		vrednost →	shrani vrednost float v lokalno spremenljivko 2
fstore_3	46		vrednost →	shrani vrednost float v lokalno spremenljivko 3
dstore_0	47		vrednost →	shrani vrednost double v lokalno spremenljivko 0
dstore_1	48		vrednost →	shrani vrednost double v lokalno spremenljivko 1
dstore_2	49		vrednost →	shrani vrednost double v lokalno spremenljivko 2
dstore_3	4A		vrednost →	shrani vrednost double v lokalno spremenljivko 3
astore_0	4B		referenca objekta →	shrani referenco v lokalno spremenljivko 0
astore_1	4C		referenca objekta →	shrani referenco v lokalno spremenljivko 1
astore_2	4D		referenca objekta →	shrani referenco v lokalno spremenljivko 2
astore_3	4E		referenca objekta →	shrani referenco v lokalno spremenljivko 3
iastore	4F		referenca polja, indeks, vrednost →	shrani vrednost int v polje
lastore	50		referenca polja, indeks, vrednost →	shrani vrednost long v polje
fastore	51		referenca polja, indeks, vrednost →	shrani vrednost float v polje
dastore	52		referenca polja, indeks, vrednost →	shrani vrednost double v polje
aastore	53		referenca polja, indeks, vrednost →	shrani referenco v polje na sklad

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
bastore	54		referenca polja, indeks, vrednost →	shrani vrednost boolean v polje
castore	55		referenca polja, indeks, vrednost →	shrani vrednost char v polje
sastore	56		referenca polja, indeks, vrednost →	shrani vrednost short v polje
pop	57		vrednost →	zavrže zgornjo vrednost sklada
pop2	58		{vrednost2, vrednost1} →	zavrže zgornji 2 vrednosti sklada
dup	59		vrednost → vrednost, vrednost	podvoji vrednost na vrhu sklada
dup_x1	5A		vrednost2, vrednost1 → vrednost1, vrednost2, vrednost1	vstavi kopijo zgornje vrednosti dve vrednosti pod vrhom
dup_x2	5B		vrednost3, vrednost2, vrednost1 → vrednost1, vrednost3, vrednost2, vrednost1	vstavi kopijo zgornje vrednosti tri vrednosti pod vrhom
dup2	5C		{vrednost2, vrednost1} → {vrednost2, vrednost1}, {vrednost2, vrednost1}	podvoji zgornji dve besedi sklada
dup2_x1	5D		vrednost3, {vrednost2, vrednost1} → {vrednost2, vrednost1}, vrednost3, {vrednost2, vrednost1}	vstavi kopijo zgornjih dveh vrednosti pod tretjo vrednost
dup2_x2	5E		{vrednost4, vrednost3}, {vrednost2, vrednost1} → {vrednost2, vrednost1}, {vrednost4, vrednost3}, {vrednost2, vrednost1}	vstavi kopijo zgornjih dveh vrednosti pod četrto vrednost
swap	5F		vrednost2, vrednost1 → vrednost1, vrednost2	zamenja zgornji dve vrednosti sklada
iadd	60		vrednost1, vrednost2 → rezultat	sešteje dve vrednosti int
ladd	61		vrednost1, vrednost2 → rezultat	sešteje dve vrednosti long
fadd	62		vrednost1, vrednost2 → rezultat	sešteje dve vrednosti float
dadd	63		vrednost1, vrednost2 → rezultat	sešteje dve vrednosti double
isub	64		vrednost1, vrednost2 → rezultat	odšteje vrednosti int
lsub	65		vrednost1, vrednost2 → rezultat	odšteje vrednosti long
fsub	66		vrednost1, vrednost2 → rezultat	odšteje vrednosti float

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
dsub	67		vrednost1, vrednost2 → rezultat	odšteje vrednosti double
imul	68		vrednost1, vrednost2 → rezultat	pomnoži vrednosti int
lmul	69		vrednost1, vrednost2 → rezultat	pomnoži vrednosti long
fmul	6A		vrednost1, vrednost2 → rezultat	pomnoži vrednosti float
dmul	6B		vrednost1, vrednost2 → rezultat	pomnoži vrednosti double
idiv	6C		vrednost1, vrednost2 → rezultat	deli vrednosti int
ldiv	6D		vrednost1, vrednost2 → rezultat	deli vrednosti long
fdiv	6E		vrednost1, vrednost2 → rezultat	deli vrednosti float
ddiv	6F		vrednost1, vrednost2 → rezultat	deli vrednosti double
irem	70		vrednost1, vrednost2 → rezultat	ostanek pri deljenju vrednosti int
lrem	71		vrednost1, vrednost2 → rezultat	ostanek pri deljenju vrednosti long
frem	72		vrednost1, vrednost2 → rezultat	ostanek pri deljenju vrednosti float
drem	73		vrednost1, vrednost2 → rezultat	ostanek pri deljenju vrednosti double
ineg	74		vrednost → rezultat	negiranje vrednosti int
lneg	75		vrednost → rezultat	negiranje vrednosti long
fneg	76		vrednost → rezultat	negiranje vrednosti float
dneg	77		vrednost → rezultat	negiranje vrednosti double
ishl	78		vrednost1, vrednost2 → rezultat	aritmetični zamik vrednosti int v levo
lshl	79		vrednost1, vrednost2 → rezultat	zamik vrednosti1 long v levo za vrednost2
ishr	7A		vrednost1, vrednost2 → rezultat	aritmetični zamik vrednosti int v desno
lshr	7B		vrednost1, vrednost2 → rezultat	zamik vrednosti1 long v desno za vrednost2
iushr	7C		vrednost1, vrednost2 → rezultat	logični zamik vrednosti int v desno
lushr	7D		vrednost1, vrednost2 → rezultat	zamik vrednosti1 long v desno za vrednost2, nepredznačeno
iand	7E		vrednost1, vrednost2 → rezultat	bitna operacija AND nad vrednostih tipa int
land	7F		vrednost1, vrednost2 → rezultat	bitna operacija AND nad vrednostih tipa long
ior	80		vrednost1, vrednost2 → rezultat	bitna operacija OR nad vrednostih tipa int
lor	81		vrednost1, vrednost2 → rezultat	bitna operacija OR nad vrednostih tipa long

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
ixor	82		vrednost1, vrednost2 → rezultat	bitna operacija XOR nad vrednostih tipa int
lxor	83		vrednost1, vrednost2 → rezultat	bitna operacija XOR nad vrednostih tipa long
iinc	84	2: indeks, konstanta	[ni sprememb]	poveča lokalno spremenljivko #indeks za konstanto
i2l	85		vrednost → rezultat	pretvori int v long
i2f	86		vrednost → rezultat	pretvori int v float
i2d	87		vrednost → rezultat	pretvori int v double
l2i	88		vrednost → rezultat	pretvori long v int
l2f	89		vrednost → rezultat	pretvori long v float
l2d	8A		vrednost → rezultat	pretvori long v double
f2i	8B		vrednost → rezultat	pretvori float v int
f2l	8C		vrednost → rezultat	pretvori flaot v long
f2d	8D		vrednost → rezultat	pretvori float v double
d2i	8E		vrednost → rezultat	pretvori double v int
d2l	8F		vrednost → rezultat	pretvori double v long
d2f	90		vrednost → rezultat	pretvori double v float
i2b	91		vrednost → rezultat	pretvori int v bajt
i2c	92		vrednost → rezultat	pretvori int v char
i2s	93		vrednost → rezultat	pretvori int v short
lcmp	94		vrednost1, vrednost2 → rezultat	primerja vrednosti tipa long
fcmpl	95		vrednost1, vrednost2 → rezultat	primerja vrednosti tipa float
fcmpg	96		vrednost1, vrednost2 → rezultat	primerja vrednosti tipa float
dcmpl	97		vrednost1, vrednost2 → rezultat	primerja vrednosti tipa double
dcmpg	98		vrednost1, vrednost2 → rezultat	primerja vrednosti tipa double
ifeq	99	2: odmik	vrednost →	če je vrednost 0, skoči na ukaz na odmiku odmik
ifne	9A	2: odmik	vrednost →	če je vrednost ni 0, skoči na ukaz na odmiku odmik
iflt	9B	2: odmik	vrednost →	če je vrednost manj kot 0, skoči na ukaz na odmiku odmik
ifge	9C	2: odmik	vrednost →	če je vrednost več ali enako 0, skoči na ukaz na odmiku odmik
ifgt	9D	2: odmik	vrednost →	če je vrednost več kot, skoči na ukaz na odmiku odmik
ifle	9E	2: odmik	vrednost →	če je vrednost manj ali enako 0, skoči na ukaz na odmiku odmik
if_icmpeq	9F	2: odmik	vrednost1, vrednost2 →	če sta vrednosti int enaki, skoči na ukaz na odmiku odmik
if_icmpne	A0	2: odmik	vrednost1, vrednost2 →	če vrednosti int nista enaki, skoči na ukaz na odmiku odmik
if_icmplt	A1	2: odmik	vrednost1, vrednost2 →	če je vrednost1 manjše od vrednost2, skoči na ukaz na odmiku odmik
if_icmpge	A2	2: odmik	vrednost1, vrednost2 →	če je vrednost1 večje ali enako vrednost2, skoči na ukaz na odmiku odmik

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
if_icmpgt	A3	2: odmik	vrednost1, vrednost2 →	če je vrednost1 večje od vrednost2, skoči na ukaz na odmiku odmik
if_icmple	A4	2: odmik	vrednost1, vrednost2 →	če je vrednost1 manjše ali enako vrednost2, skoči na ukaz na odmiku odmik
if_acmpeq	A5	2: odmik	vrednost1, vrednost2 →	če sta referenci enaki, skoči na ukaz na odmiku odmik
if_acmpne	A6	2: odmik	vrednost1, vrednost2 →	če referenci nista enaki, skoči na ukaz na odmiku odmik
goto	A7	2: odmik	[ni sprememb]	skoči na ukaz na odmiku odmik
jsr	A8	2: odmik	→ naslov	skoči na podprogram na odmiku odmik
ret	A9	1: indeks	[ni sprememb]	nadaljuj z izvajanjem na naslovu iz lokalne spremenljivke #indeks
tableswitch	AA	4+: [0-3B zapolnjevanja], p1, p2, p3, p4, s1, s2, s3, s4, z1, z2, z3, z4, odmiki …	indeks →	nadaljuj z izvajanjem na naslovu v tabeli z odmikom indeks
lookupswitch	AB	4+: [0-3B zapolnjevanja], p1, p2, p3, p4, np1, np2, np3, np4, odmiki ...	ključ →	naslov se prebere iz tabele z uporabo ključa in izvajanje se nadaljuje na tem naslovu
ireturn	AC		vrednost → [prazno]	vrne vrednost int iz metode
lreturn	AD		vrednost → [prazno]	vrne vrednost long iz metode
freturn	AE		vrednost → [prazno]	vrne vrednost float iz metode
dreturn	AF		vrednost → [prazno]	vrne vrednost double iz metode
areturn	B0		referenca objekta → [prazno]	vrne referenco iz metode
return	B1		→ [prazno]	vrne void iz metode
getstatic	B2	2: indeks	→ vrednost	naloži vrednost statičnega polja razreda, kjer je polje definirano v tabeli konstant z oznako indeks
putstatic	B3	2: indeks	vrednost →	shrani vrednost v statično polje razreda, kjer je polje definirano v tabeli konstant z oznako indeks
getfield	B4	2: indeks	referenca objekta → vrednost	naloži vrednost polja objekta referenca, kjer je polje definirano v tabeli konstant z oznako indeks
putfield	B5	2: indeks	referenca objekta, vrednost →	shrani vrednost v polje objekta referenca, kjer je polje definirano v tabeli konstant z oznako indeks
invokevirtual	B6	2: indeks	referenca objekta, [arg1, arg2, ...] →	pokliči metodo objekta referenca, kjer je metoda definirana v tabeli konstant z oznako indeks
invokespecial	B7	2: indeks	referenca objekta, [arg1, arg2, ...] →	pokliči metodo objekta referenca, kjer je metoda definirana v tabeli konstant z oznako indeks
invokestatic	B8	2: indeks	[arg1, arg2, ...] →	pokliči statično metodo, kjer je metoda definirana v tabeli konstant z oznako indeks
invokeinterface	B9	4: indeks1, indeks2, število, 0	referenca objekta, [arg1, arg2, ...] →	pokliči metodo vmesnika referenca, kjer je metoda definirana v tabeli konstant z oznako indeks
invokedynamic	BA	4: indeks1, indeks2, 0, 0	[arg1, [arg2 ...]] →	pokliči dinamično metodo definirano v tabeli konstant z oznako indeks
new	BB	2: indeks	→ referenca objekta	ustvari nov objekt tipa definiranega v tabeli konstant z oznako indeks
newarray	BC	1: tip	število → referenca polja	ustvari novo polje primitivnega tipa velikosti število
anewarray	BD	2: indeks	število → referenca polja	ustvari novo polje referenc velikosti število, kjer je referenca definirana v tabeli konstant z oznako indeks

Ukaz	#	Velikost: pomen operandov	Sprememba sklada	Opis ukaza
arraylength	BE		referenca polja → dolžina	prebere velikost polja referenca
athrow	BF		referenca objekta → [prazno], referenca objekta	povzroči izjemo ali napako
checkcast	C0	2: indeks	referenca objekta → referenca objekta	preveri če je referenca tipa definiranega v tabeli konstant z oznako indeks
instanceof	C1	2: indeks	referenca objekta → rezultat	preveri če je referenca primerek tipa definiranega v tabeli konstant z oznako indeks
monitorenter	C2		referenca objekta →	vstop v monitor objekta
monitorexit	C3		referenca objekta →	izstop iz monitorja objekta
wide	C4	3: opcode, i1, i2 ALI 5: 84, i1, i2, c1, c2	[isto kot pri ustreznih ukazih]	izvrši ukaz op za nalaganje s 16 bitnim naslovom ali ukaz iinc z dvema 16 bitnima operandoma
multianewarray	C5	3: indeks1, indeks2, dimenzije	števec1, [števec2,...] → referenca polja	ustvari novo polje dimenzij dimenzije z elementi tipa definiranega v tabeli konstant z oznako indeks
ifnull	C6	2: odmik	vrednost →	če je vrednost null, skoči na ukaz na odmiku odmik
ifnonnull	C7	2: odmik	vrednost →	če vrednost ni null, skoči na ukaz na odmiku odmik
goto_w	C8	4: odmik	[ni sprememb]	skoči na ukaz na odmiku odmik
jsr_w	C9	4: odmik	→ naslov	skoči na podprogram na odmiku odmik
breakpoint	CA			rezervirano za ustavitvene točke, se ne pojavlja v običajnih datotekah class
(ni imena)	CB – FD			nedefinirani ukazi, rezervirani za prihodnjo uporabo
impdep1	FE			rezervirano, se ne pojavlja v običajnih datotekah class
impdep2	FF			rezervirano, se ne pojavlja v običajnih datotekah class

Poglavje 3 Zahteve razširjenega prevajalnika

Cilj izdelave razširjenega prevajalnika je osnovna podpora programiranju z zložno kodo. Odločili smo se, da podobno kot v programskem jeziku C, uporabimo nov ukaz `jasm`. Ta ukaz naznanja blok, ki vsebuje ukaze zložne kode. Glavna naloga prevajalnika je določiti mesto uporabljenih `jasm` blokov znotraj programa, ki je napisan v Javi ter vriniti ukaze zložne kode v prevedeno datoteko. Prevajalnik naj bi bil podprt na vseh sistemih in deloval na vseh različicah Jave. Zaradi lažje izvedbe smo se odločili tudi za nekaj omejitev, ki so opisane v nadaljevanju.

3.1 Jasm blok

Razširjeni prevajalnik za Javo naj bi, poleg standardnih javanskih ukazov, podpiral tudi bloke z ukazi zložne kode. Odločili smo se za uporabo novega ukaza `jasm`, ki mu sledijo ukazi zložne kode v zavitem oklepaju. Znotraj posameznega bloka so ukazi zložne kode ločeni s podpičji, operandi pa so od ukazov in med seboj ločeni s presledki. Da bi bilo programiranje uporabniku bolj prijazno in lažje razumljivo, smo pri operandih ukazov zložne kode dopustili uporabo števil v desetiškem zapisu ter imena deklariranih javanskih spremenljivk. Uporabniku torej ni treba pretvarjati želenih vrednosti v dvojiški zapis ali v ustrezen format glede na uporabljen ukaz. Prevajalnik pretvori vsa števila v ustrezen format in s pomočjo tabele lokalnih spremenljivk poskrbi, da se vrednosti zapišejo na ustrezno mesto oziroma se iz ustreznega mesta preberejo. Spodaj sta dva primera preprostega programa, napisana v Javi. V tabeli Tabela 9 so uporabljeni zgolj javanski ukazi, v tabeli Tabela 10 pa je prikazana uporaba `jasm` bloka. Rezultat obeh programov je enak.

```
public class Test {
	public static void main(String[] args) {
		int a = 1;
		int b = 6;
		int test = a + b;
		System.out.println(test);
	}
}
```

Tabela 9 - Enostaven primer programa v Javi.

```
public class Test {
	public static void main(String[] args) {
		int a = 1;
		int b = 6;
		int test = 0;
		jasm {
			iload a;
			iload b;
			iadd;
			istore test;
		}
		System.out.println(test);
	}
}
```

Tabela 10 - Prikaz uporabe `jasm` bloka med vrsticami javanske kode.

Iz primera v tabeli Tabela 10 je razviden način uporabe `jasm` bloka. V tej kodi je vsebina bloka oblikovana v stilu javanske kode. Nove vrstice za ukazi niso potrebne, je pa koda tako preglednejša. Dodamo lahko še to, da se lahko znotraj posameznih metod uporabi poljubno število `jasm` blokov.

3.2 Združljivost

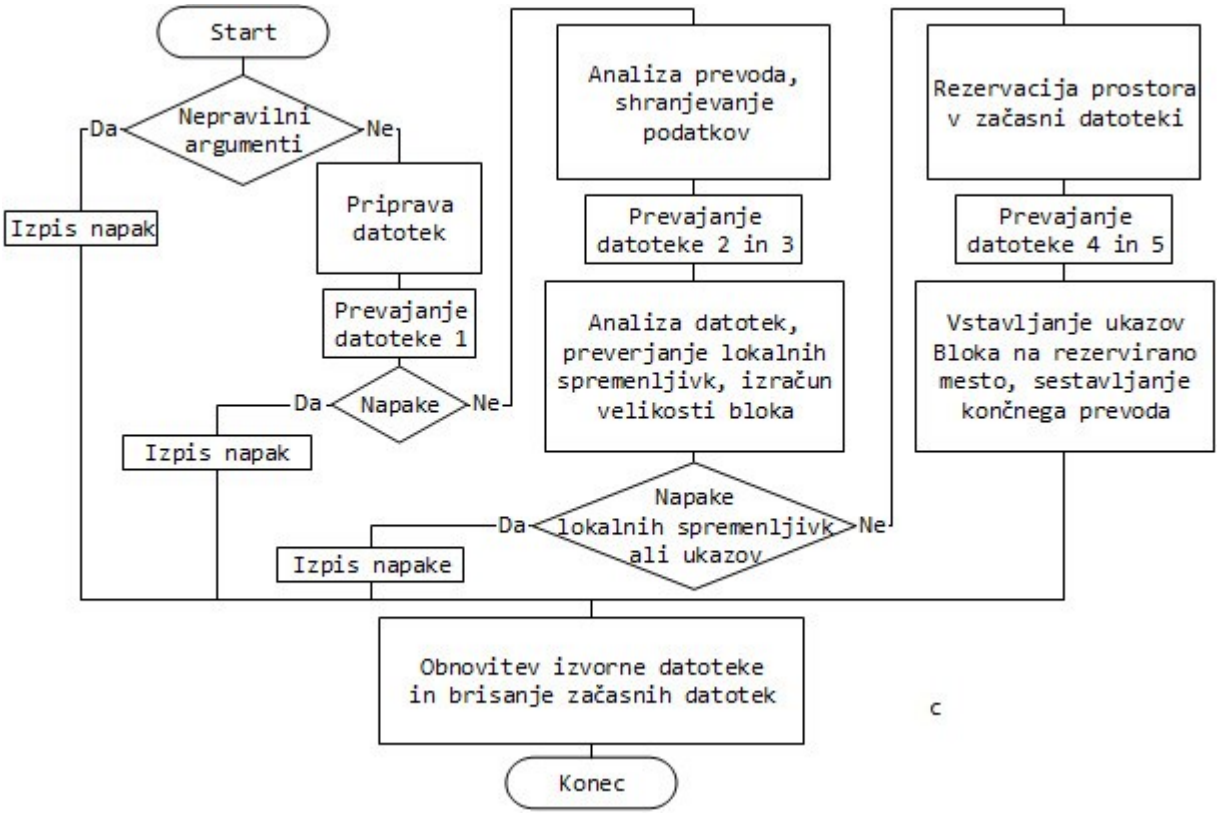
Pri izdelavi prevajalnika smo za enega izmed ciljev določili tudi delovanje prevajalnika na vseh različicah Jave, kot tudi na vseh sistemih. Ob prvem razmisleku k slednjemu pripomore že sama izbira Jave kot jezika za razvoj prevajalnika. Kot smo že omenili, je Java sistemsko neodvisna in programi napisani v Javi delujejo na vseh sistemih, ki imajo naložen ustrezen JVM. Za delovanje na starejših različicah Jave se je treba izogniti javanskim ukazom, ki v njih niso na voljo. Več o doseganju združljivosti je zapisanega v naslednjem poglavju.

3.3 Omejitve prevajalnika

Zaradi preprostosti prevajalnika smo se odločili, da za zdaj ne bomo podprli vpisovanja novih zapisov v tabelo konstant. To pomeni, da mora programer uporabljati samo spremenljivke, ki so že bile deklarirane pred uporabo `jasm` bloka. Enako velja za vrednosti znakovnih konstant, kot tudi za vrednosti številskih konstant, kjer velikost števila ni predstavljiva s številom bitov, ki so nam na voljo za operand pri posameznem ukazu. Med možnostmi za razširitve so opisane težave, ki bi jih povzročilo dodajanje novih spremenljivk.

Poglavje 4 Razvoj prevajalnika

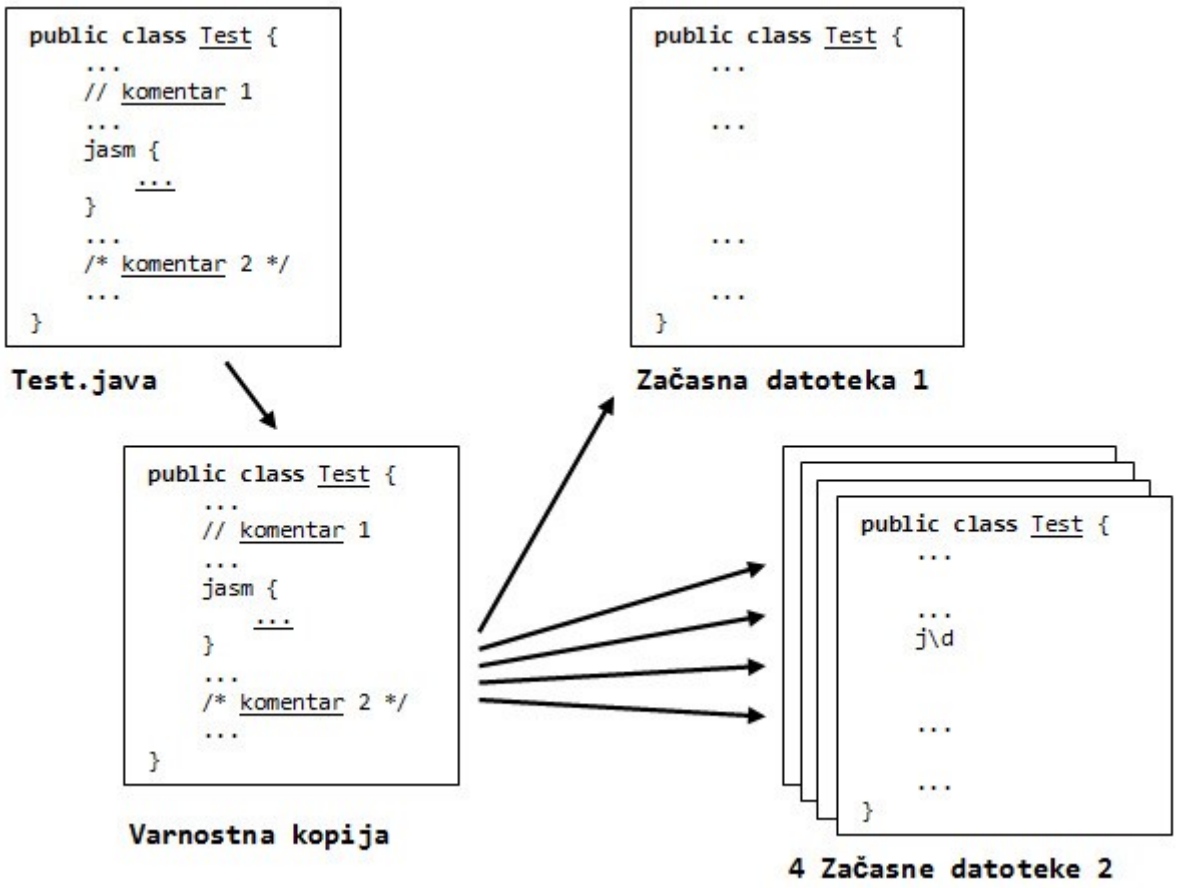
Izvedba samega prevajalnika je precej preprosta, ko se enkrat dodobra spoznamo s sestavo prevedenih datotek. Najprej je treba iz programa izluščiti `jasm` bloke. Nato se večina dela skriva v tem, da analiziramo prevedene datoteke, ki so bile tako ali drugače spremenjene. A kljub vsemu je treba biti pozoren na določene stvari, ki se ob prvotnem razmišljanju o rešitvi ne zdijo težavne. Včasih se izkaže, da se je izvedbe treba lotiti drugače. V nadaljevanju si bomo podrobno ogledali razvoj prevajalnika, na kakšne težave smo naleteli ter kako smo jih rešili. Slika 1 prikazuje diagram poteka razvitega prevajalnika, ki je v grobem sestavljen iz petih glavnih delov: priprava datotek, analiza prevedene datoteke, določitev mesta ter analiza kode, rezervacija prostora in vstavljanje kode.



Slika 1 - Diagram poteka za razširjeni prevajalnik.

4.1 Priprava datotek

Kot smo že omenili, bo za delovanje prevajalnika potrebnih več sprememb izvorne datoteke. Uporabnik navadno pričakuje, da po prevodu njegova koda ostane nespremenjena, zato si najprej naredimo varnostno kopijo izvorne datoteke. To naredimo tako, da datoteko preimenujemo, ker bomo kasneje iz nje kopirali ustrezno vsebino. Ime izvorne datoteke bomo potrebovali pri prevajanju z navadnim javanskim prevajalnikom, kjer mora biti ime datoteke enako imenu razreda. Sledi branje preimenovane izvorne datoteke, ki jo prepisujemo v dve začasni datoteki. Prepisujemo samo javansko kodo in spuščamo vse vrste komentarjev. Če naletimo na `jasm` blok, si njegovo vsebino shranimo v seznam, kjer hranimo informacijo o zaporedni številki bloka in vsebino. Vsebino si shranimo skupaj s številkami vrstic, če gre za večvrstični blok. To nam omogoča, da kasneje pri opozorilu o napaki izpišemo tudi, kje je do napake prišlo. Če je bilo uporabljeno zaporedje znakov `jasm` kot ime spremenljivke ali metode in mu ne sledi blok z zložno kodo v zavitih oklepajih, moramo uporabnika opozoriti, da je pri uporabi tega prevajalnika to rezervirana beseda in je ne more uporabljati kot ime spremenljivke ali metode.

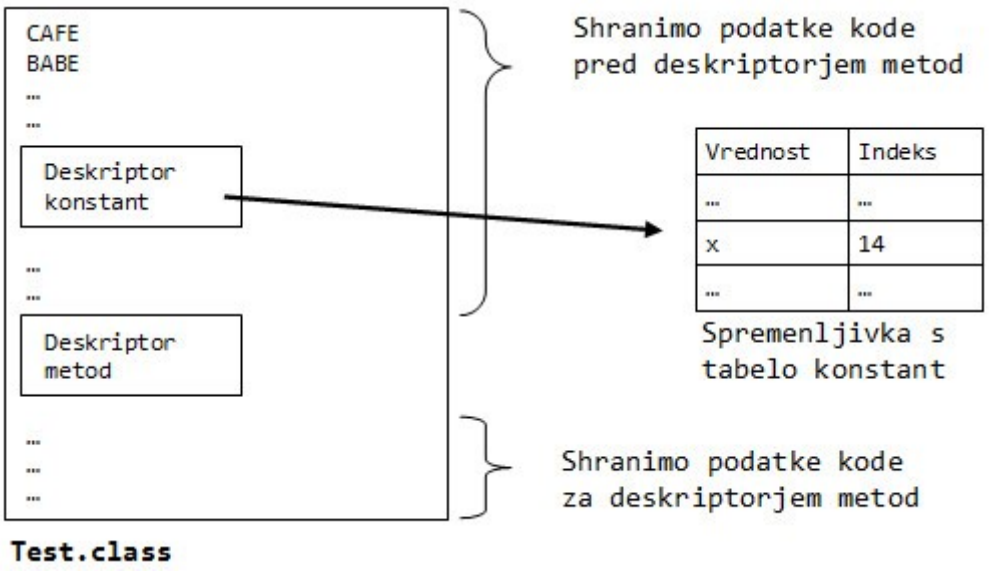


Slika 2 - Grafični prikaz priprave datotek.

Na sliki Slika 2 je grafično prikazan postopek priprave datotek. V prvi začasni datoteki `jasm` blok v celoti izpustimo, v drugi pa si označimo njegovo mesto z zaporedjem znakov. Ker v javanski kodi še vedno obstajajo znakovne konstante, moramo zaporedje znakov izbrati tako, da to zaporedje tvori neveljaven znakovni niz. Sicer bi se lahko zgodilo, da bi pri naslednjih menjavah izbranega zaporedja znakov spremenili vsebino znakovne konstante in s tem delovanje programa. V našem primeru smo si za takšno zaporedje izbrali znakovno zaporedje `j\d`. Če uporabnik to zaporedje uporabi v znakovni konstanti, nas običajni prevajalnik opozori o napaki, saj je `\d` neveljavna ubežna sekvenca. Črka pred njo nam zagotovi, da pred `\` ni znaka `\`, ki pa skupaj tvorita veljavno ubežno sekvenco. Ko zaključimo z analizo izvorne datoteke, si ustvarimo še tri kopije začasne datoteke z oznako mest blokov, ki jih bomo potrebovali v naslednjih korakih.

4.2 Analiza prevedene datoteke

Vzamemo začasno datoteko brez zaporedij znakov, ki označujejo mesta `jasm` blokov. To datoteko nato preimenujemo v izvorno ime in jo prevedemo z javanskim prevajalnikom. Če je med prevajanjem prišlo do napake, rezultat izpišemo na zaslon in končamo z delom. V nasprotnem primeru dobimo kot rezultat delujočo datoteko, ki pa ne vsebuje ukazov zložne kode iz `jasm` blokov.



Slika 3 - Grafični prikaz analize prevedene datoteke.

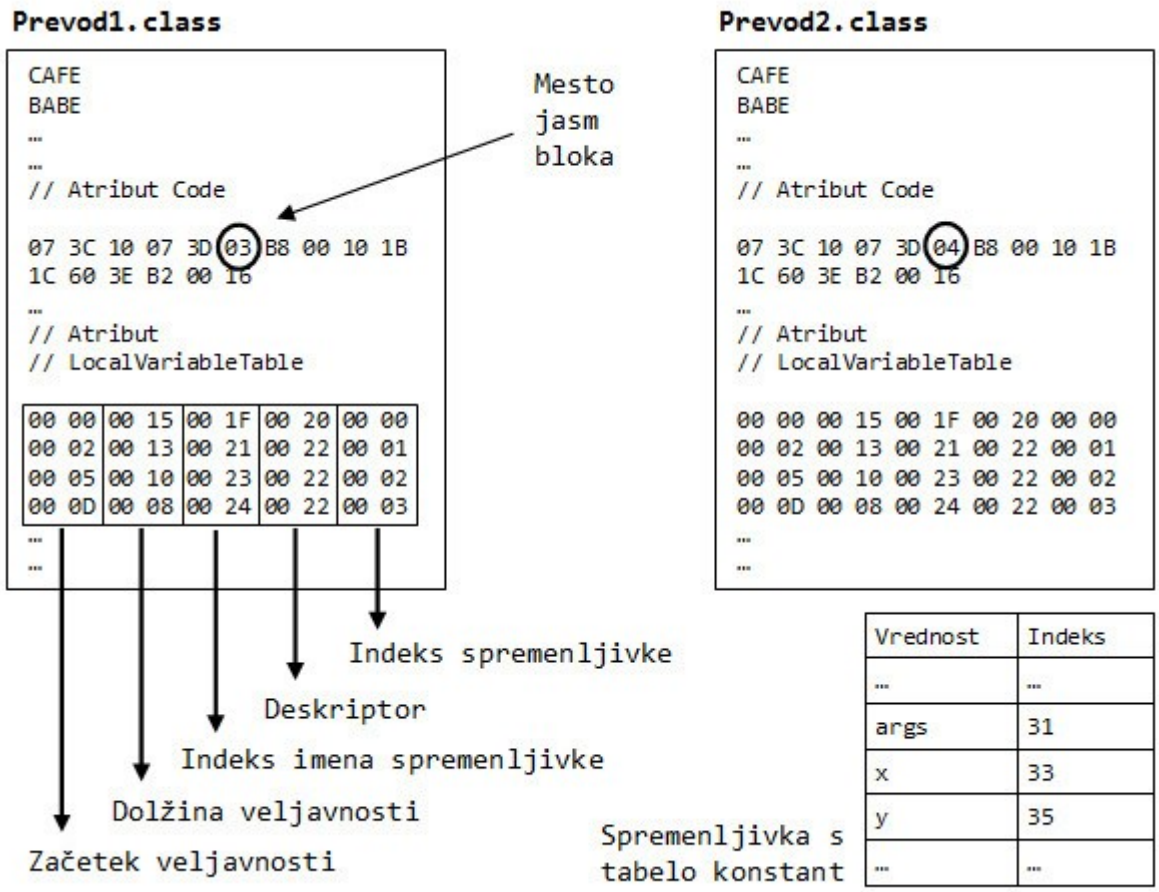
Kodo prevajamo z dodatnim parametrom `-g`, kjer prevajalnik v prevedeni datoteki doda podatke za razhroščevanje. Sem sodijo zapisi spremenljivk v tabeli konstant kot tudi tabele

lokalnih spremenljivk posameznih metod. Ta datoteka nam služi kot iztočnica, saj poleg podatkov za razhroščevanje nima nobene dodatne vsebine, ki je potrebna za delovanje razširjenega prevajalnika. Dobljeno prevedeno datoteko analiziramo in si shranimo celotno vsebino pred in po deskriptorju metod, ki je edini del, ki ga bomo spremenili in zamenjali. Poleg tega si shranimo tudi tabelo konstant, tako da je z vsebino konstante mogoče dobiti njeno zaporedno številko, kot je to prikazano na sliki Slika 3. V vsebini konstant so namreč shranjena tudi imena lokalnih spremenljivk in metod, ki jih programer lahko uporablja kot atribute pri ukazih zložne kode.

4.3 Določitev mesta in analiza kode

Sledi tretji korak prevajanja, ki je od vseh najbolj zahteven, saj v njem naredimo tri stvari hkrati. Prva stvar je določanje mesta, kjer se `jasm` blok nahaja. Kot smo videli v drugem poglavju je nekaj kod za ukaze nezasedenih. Poleg tega, obstajata tudi kodi za ustavitvene točke (*angl. break point*) in operacijo NOP (*angl. no operation*), ki se sicer v običajnih prevodih ne pojavljata. S temi kodami bi si lahko pomagali določiti mesto `jasm` bloka znotraj ukazov prevedene metode. Toda težava nastane pri vprašanju, kako z običajnim prevajalnikom te ukaze sploh zapisati v prevedeno datoteko. Poleg tega bi v tem primeru morali ob analizi vedno preverjati kateri bajt predstavlja ukaz in kateri atribut ali le del njih. Zato smo prišli na drugačno idejo. Mesto `jasm` bloka določimo s pomočjo dveh kopij začasne datoteke, kjer zamenjamo znakovni niz, ki označuje mesto `jasm` bloka v javanski kodi. V prvi datoteki niz zamenjamo z ukazom `System.exit(0)`, v drugem pa z ukazom `System.exit(1)`. Dobra lastnost teh dveh ukazov je v tem, da se prevoda razlikujeta že v prvem bajtu, ker najprej nastopi ukaz, ki parameter naloži na sklad. Seveda je kot parameter treba izbrati številski konstanti, ki imata skrajšana ukaza, sicer bi bilo določanje mesta bolj zapleteno. Datoteki z zamenjano vsebino nato prevedemo in ju vzporedno analiziramo. Kjer pride do spremembe v deskriptorju metod je mesto, kamor je treba vstaviti ukaze `jasm` bloka. Če je `jasm` blokov več, nadaljujemo do naslednje spremembe, kjer je naslednji blok, saj je prevajanje zaporedno in si mesta blokov sledijo enako kot njihove pojavitve v javanski kodi.

Ker je analiza datoteke časovno potraten postopek, se želimo izogniti odvečnim analizam. Po analiziranju kode posamezne metode se nahajamo v atributih deskriptorja metod, kjer je zapisana tudi tabela lokalnih spremenljivk. Zato je najbolj primerno, da v tem koraku preverimo tudi vidljivost lokalnih spremenljivk, kar je prikazano na sliki Slika 4. Za vse ukaze ustreznega bloka preverimo operande ukazov, ki uporabljajo lokalne spremenljivke. Iz tabele spremenljivk nato preberemo ali je uporabljena spremenljivka vidna na mestu obravnavanega `jasm` bloka. V kolikor je lokacija bloka izven obsega, ki označuje vidljivost



Slika 4 - Preverjanje vidljivosti lokalnih spremenljivk.

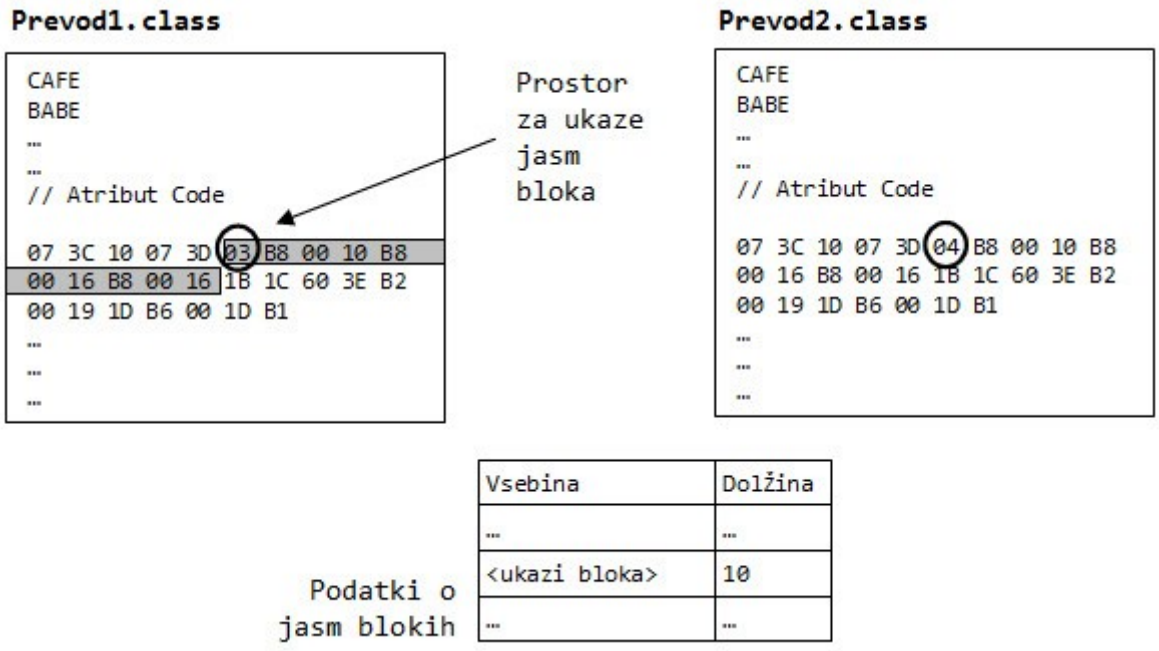
spremenljivke, na to opozorimo uporabnika. Na tem mestu uredimo tudi atribute drugih ukazov in jih pretvorimo v ustrezno obliko. S tem se izognemo eni odvečni analizi v primeru, da je med ukazi vrinjene zložne kode prišlo do napake. Uporabnika na napako opozorimo že v tem koraku in končamo z delom, saj delujočega prevoda ni mogoče narediti. Pri morebitnih klicih funkcij ali uporabi znakovnih konstant uporabimo shranjeno tabelo konstant iz prve datoteke. To je pomembno zato, ker nam vstavljanje ukaza `System.exit` v trenutno tabelo konstant lahko spremeni vrstni red indeksov glede na prvi prevod. Pri sestavljanju končne prevedene datoteke bomo uporabili kodo pred deskriptorjem metod iz prve datoteke, kjer je tudi tabela konstant. Če bi uporabili napačno tabelo konstant, bi lahko prišlo do napačnega delovanja programa ali program celo ne bi deloval.

Tretja stvar, ki jo naredimo v tem koraku, je izračun velikosti prostora, ki ga potrebujejo ukazi zložne kode v posameznem `jasm` bloku. To bi sicer lahko naredili že takoj po branju `jasm` bloka, a je izračun na tem mestu optimalnejši. V tem koraku lahko namreč zamenjamo vse

lokalne spremenljivke z ustreznimi indeksi iz tabele lokalnih spremenljivk. Nato lahko nekatere ukaze z znanimi operandi zamenjamo z njihovimi skrajšanimi različicami. Tako lahko namesto ukaza `iload` z operandom `1`, ki zasede dva bajta, uporabimo ukaz `iload_1` velikosti en bajt. S tem dobimo bolj optimalno prevedeno datoteko.

4.4 Rezervacija prostora

Sprva se je vrivanje dodatnih ukazov zložne kode zdelo precej preprosto. Ideja je bila, vriniti ukaze na mesto, ki smo ga izračunali v prejšnjem koraku. Poleg tega bi bilo treba še povečati števec, ki v deskriptorju označuje velikost atributa s kodo. Toda izkaže se, da bi ta rešitev delovala zgolj za najbolj preproste programe. Gre za programe, ki ne uporabljajo skokov, kar pomeni za programe brez zank in pogojnih stavkov. Težava je namreč v dejstvu, da Java v prevedenih datotekah uporablja relativne skoke. Ker v datotekah z zložno kodo ni nobenih oznak z naslovi, so skoki definirani s fiksno številko, ki nam pove, koliko bajtov naprej ali nazaj nadaljujemo z izvajanjem. To nam seveda podre našo prvotno idejo, saj bi v primeru uporabe `jasm` bloka znotraj zanke, bili odmiki pri skokih napačni. Lahko bi sicer poskušali najti vse skoke in ustrezno popravili njihove odmike, a bi bilo to precej zamudno delo. Za vsak skok bi bilo treba ugotoviti ali `jasm` blok sploh vpliva na njegov odmik ter ga nato ustrezno povečati oziroma zmanjšati. Izkaže se, da je preprostejše, če si prostor potreben za dodatne ukaze vnaprej rezerviramo.



Slika 5 - Prikaz rezervacije prostora in ukazov, ki se v naslednjem koraku zamenjajo.

Kot je prikazano na sliki Slika 5, rezervacijo naredimo s pomočjo preostalih dveh datotek, kjer izbrani niz, ki označuje mesto `jasm` bloka, zamenjamo z javanskimi ukazi. V prvi datoteki zopet uporabimo ukaz `System.exit(0)`, v drugi pa `System.exit(1)`, ki zasedeta po štiri bajte. Za tema ukazoma uporabimo ukaze `System.gc()`. Ta ukaz se prevede v tri bajte in ne dodaja novih zapisov v tabelo lokalnih spremenljivk. Uporabimo toliko klicev `System.gc()`, da velikost prevoda ustreza številu bajtov, potrebnih za ukaze `jasm` bloka. Spremenjeni datoteki zopet prevedemo in dobimo prevedeni datoteki z rezerviranim prostorom. Mesta blokov smo v prejšnjem koraku sicer že izračunali, a bi bilo treba te številke pri vsakem bloku spremeniti glede na velikosti vseh predhodnih blokov. Zato je bolj preprosto, da uporabimo dve datoteki in za določanje mesta vrinjenih ukazov uporabimo isti trik kot v prejšnjem koraku.

4.5 Vstavljanje kode

Preostane nam še zadnji korak prevajanja. V tem koraku uporabimo prevedeni datoteki z rezerviranim prostorom in ju zopet vzporedno analiziramo. Kjer v atributu s kodo metode pride do razlike, začnemo z vstavljanjem ukazov bloka. Poleg tega si celoten deskriptor metod shranimo, saj je to edini del, ki ga bomo potrebovali. Za konec nam ostane še manjša podrobnost, in sicer je treba popraviti tabele lokalnih spremenljivk za vsako metodo. Treba je zamenjati indekse imen lokalnih spremenljivk, ki v tem primeru kažejo na tabelo konstant iz trenutne datoteke. Če bi uporabili tabelo lokalnih spremenljivk iz prvega prevoda, bi imeli napačne podatke o vidljivosti spremenljivk zaradi vrinjenih ukazov. Naj še enkrat omenimo, da je tabela lokalnih spremenljivk namenjena zgolj razhroščevanju in bi program deloval tudi brez popravka. Toda ker ne vemo, kaj bo uporabnik s prevedeno datoteko počel, je pravilno, da prevod ne vsebuje takšnih neskladij. Končno prevedeno datoteko sestavimo tako, da uporabimo kodo pred in po deskriptorju metod iz prve analize, vmes pa dodamo popravljeni deskriptor metod.

4.6 Doseganje združljivosti

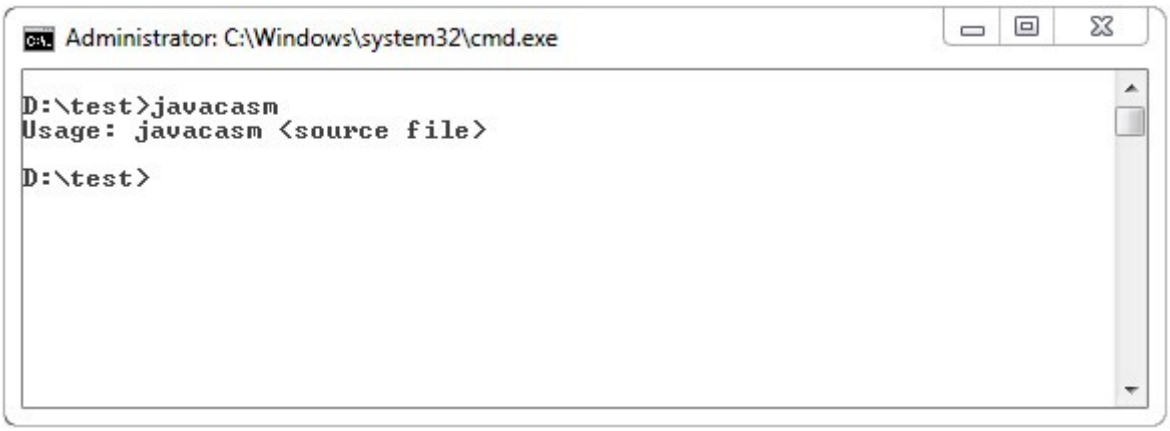
Za enega izmed ciljev razširjenega prevajalnika smo določili tudi delovanje na vseh sistemih, kot tudi na vseh različicah Jave. Kot se je izkazalo, obojega hkrati žal ni mogoče doseči. Težava je v uporabi knjižnic za prevajanje datotek z običajnim prevajalnikom. Med razvojem smo uporabili tri različne pristope k prevajanju datotek, kjer ima vsak svoje prednosti in svoje slabosti. Najprej smo se prevajanja lotili z uporabo knjižnice `Runtime`, kjer lahko s pomočjo metode `getRuntime` sistemu pošljemo poljuben ukaz, kot bi to storili v ukazni vrstici. Dobra lastnost tega načina je razpoložljivost knjižnice že od prve različice Jave. Toda tu se pojavi

vprašanje, ali za prevajanje javanskih datotek res vsi sistemi uporabljajo isti ukaz, saj bi v nasprotnem primeru za delovanje prevajalnika bilo treba poseči v izvorno kodo. Ob iskanju bolj integrirane rešitve, smo naleteli na knjižnico `JavaCompiler`, kjer nam metoda `getCompiler` vrne prevajalnik iz trenutno naloženega JDK. Če JDK ni naložen, nam metoda vrne vrednost `null` in prevajalnik ne deluje. A kot smo že ugotovili, bo uporabnik razširjenega prevajalnika zagotovo imel naložen JDK. Poleg tega, da je omenjena knjižnica na voljo šele od različice 1.6, se nam pri obeh opisanih metodah pojavlja še eno vprašanje. Kot smo zapisali že v uvodu, prevajanje v zložno kodo ni natančno določeno, mi pa se v našem prevajalniku zanašamo, da se ukazi `System.exit(0)`, `System.exit(1)` ter `System.gc()` prevedejo v točno določeno zaporedje ukazov zložne kode. Odločili smo se še naprej iskati možnosti za izboljšavo prevajanja in preizkusili prevajalnik od orodja eclipse, ki je v najnovejši različici dostopen kot samostojna komponenta na njihovi spletni strani [8]. Ugotovili smo, da tudi ta prevajalnik za delovanje zahteva vsaj različico 1.6, deluje pa tudi na sistemih, ki imajo naložen zgolj JRE. Uporaba takšnega prevajalnika nam zagotavlja, da se javanski ukazi vedno prevedejo v isto zaporedje ukazov zložne kode, vendar pa ima uporaba takšnega prevajalnika drugo slabost. Ob vsaki novi različici Jave bi bilo treba obstoječi prevajalnik zamenjati z novejšo različico. Na koncu smo se odločili za uporabo knjižnice `JavaCompiler`, saj je njena uporaba še najbolj preprosta. V razširjenem prevajalniku so ostale vse tri metode za prevajanje datotek in uporaba želene metode se po potrebi lahko zamenja.

Poglavje 5 Delovanje in uporaba prevajalnika

Prevajalnik je objavljen na spletu kot javni github repozitorij [9]. Za uporabo je treba imeti nameščen JDK, kar se pričakuje od nekoga, ki želi programirati v Javi. Uporaba je precej preprosta in simulira uporabo običajnega javanskega prevajalnika. Za operacijski sistem Windows je priložena izvajalna datoteka `javacasm`, ki se jo s prevedeno kodo prevajalnika prekopira v poljubno mapo. Najlažje je, da je pot do mape zapisana v spremenljivki okolja `PATH`. To nam omogoča, da v ukazni vrstici uporabimo ukaz `javacasm <ime datoteke>`, ne glede na to, kje se nahajamo. Pri drugih sistemih je treba ustrezno bližnjico še izdelati, lahko pa program preprosto izvajamo kot vsak drug program napisan v Javi. Pri tem kot parameter uporabimo ime datoteke, ki jo želimo prevesti. V nadaljevanju je prikazana uporaba izdelanega prevajalnika na testnem primeru. Testni primer vsebuje različne napake, ki jih postopoma odpravljamo. Vidimo lahko, na kakšne napake nas prevajalnik opozori in kakšna so opozorila. Ker razširjeni prevajalnik uporablja standardni javanski prevajalnik in je namenjen širši javnosti, so sporočila o napakah v angleškem jeziku.

Kot smo že povedali, razširjeni prevajalnik kot parameter prejme ime datoteke, ki jo želimo prevesti. V kolikor na to pozabimo, nas prevajalnik o tem obvesti, kot prikazuje Slika 6.



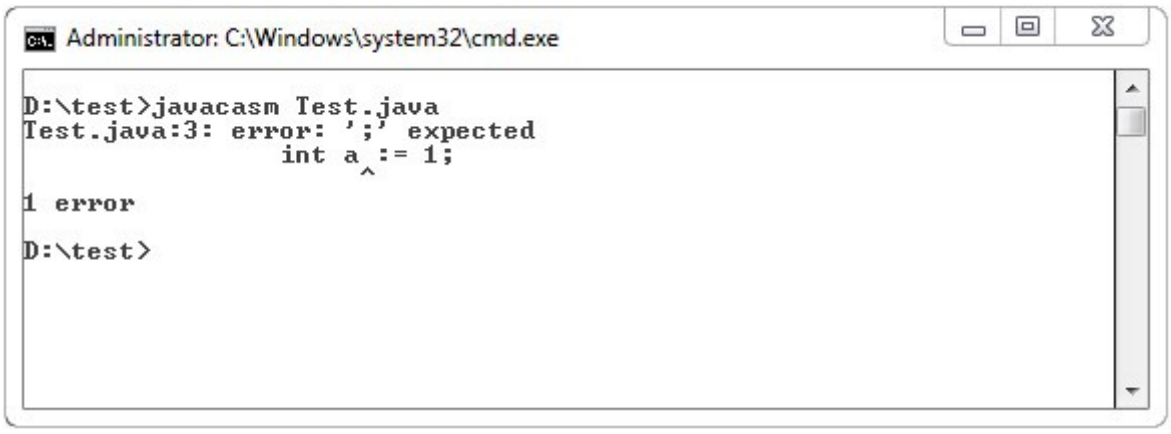
Slika 6 - Klic prevajalnika brez parametra.

Testni primer `Test.java` je prikazan v tabeli Tabela 11. V primeru smo želeli uporabiti `jasm` blok in preveriti delovanje prevajalnika. Vrinjena koda je v zanki, s čimer smo preverili delovanje skokov pri uporabi `jasm` bloka.

```
public class Test {
	public static void main(String[] args) {
		int a := 1;
		for(int i=0; i<10; i++) {
			jasm {
				iload b;
				bipush 1;
				iadd a;
				istore c;
			}
		}
		int b = 7;
		System.out.println(a*b);
	}
}
```

Tabela 11 - Testni primer z napakami.

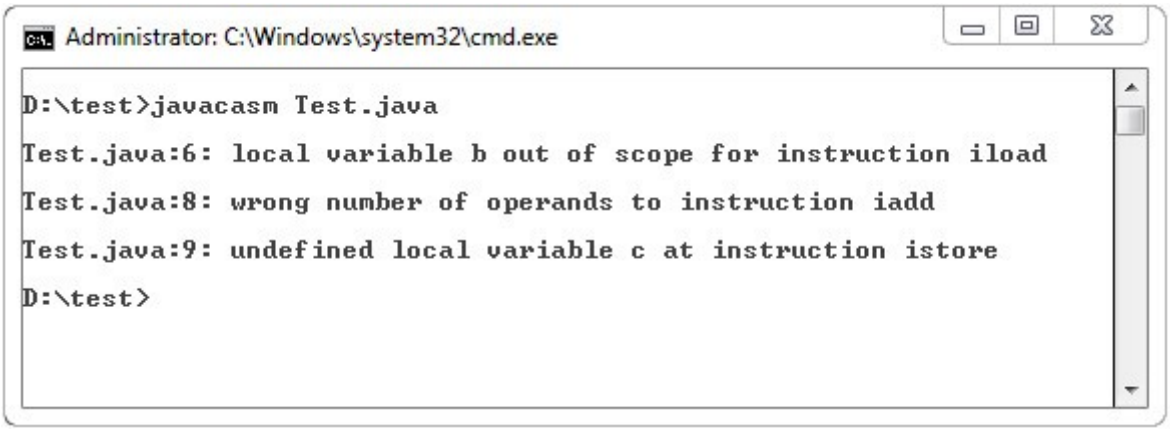
Na sliki Slika 7 vidimo opozorilo posredovano od običajnega javanskega prevajalnika. Pri pisanju primera je prišlo do napake v tretji vrstici, saj je `int a := 1;` nepravilna inicializacija v programskem jeziku Java.



Slika 7 - Izpis napake običajnega prevajalnika.

Ko iz programa odstranimo odvečni znak `:` in javanski del kode ne vsebuje več napak, nas razširjeni prevajalnik opozori na morebitne napake znotraj `jasm` blokov. V našem primeru

imamo tri napake. V šesti vrstici je uporabljena lokalna spremenljivka `b`, ki na tem mestu še ni vidna. V osmi vrstici je ukaz `iadd` zapisan z odvečnim operandom, v deveti pa je uporabljena lokalna spremenljivka `c`, ki sploh ne obstaja. Razširjeni prevajalnik nas obvesti o vseh napakah, kot to prikazuje Slika 8.

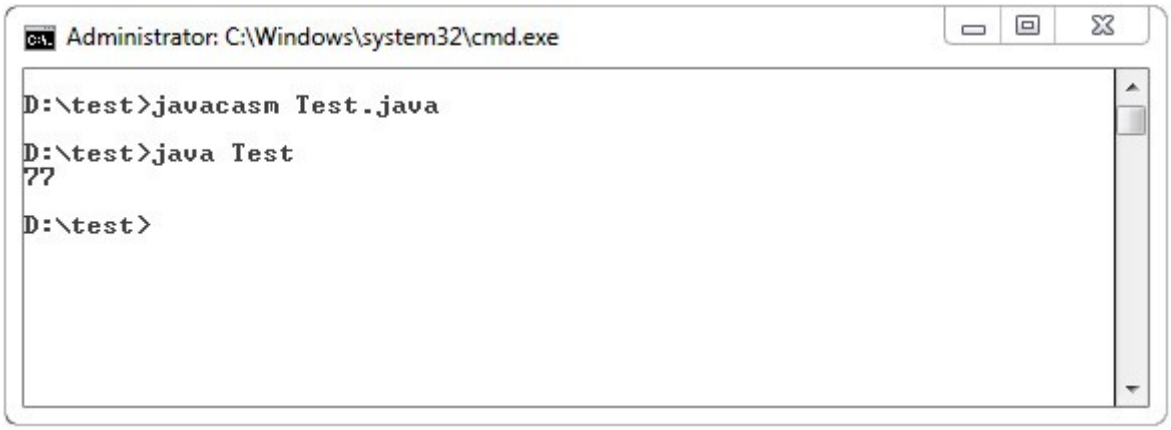


Slika 8 - Izpis napak med ukazi zložne kode.

Ko odpravimo tudi napake znotraj `jasm` blokov, dobimo delujočo javansko kodo z vrinjeno zložno kodo. Koda delujočega testnega primera je prikazana v tabeli Tabela 12, uspešno preveden in zagnan program pa prikazuje Slika 9.

```
public class Test {
	public static void main(String[] args) {
		int a = 1;
		for(int i=0; i<10; i++) {
			jasm {
				iload a;
				bipush 1;
				iadd;
				istore a;
			}
		}
		int b = 7;
		System.out.println(a*b);
	}
}
```

Tabela 12 - Delujoč testni primer.



Slika 9 - Uspešno preveden in zagnan testni primer.

Poglavje 6 Sklepne ugotovitve

Izdelali smo delujoč prevajalnik, ki po večini ustreza opisanim zahtevam. Žal smo ugotovili, da združljivost za vse sisteme, kot tudi za vse verzije Jave hkrati, ni mogoče doseči na preprost način. Odločili smo se, da je sistemska neodvisnost pomembnejša, kot delovanje na starejših različicah Jave. Poleg že opisanih omejitev, pa se je med samim razvojem porodilo veliko dodatnih idej za izboljšavo in nadgradnjo prevajalnika, ki bi jih v prihodnosti lahko uresničili. V nadaljevanju so opisane nekatere možnosti za nadgradnjo obstoječega prevajalnika.

6.1 Optimizacija končne datoteke in dodatni parametri

Nadaljnjo optimizacijo izhodne datoteke bi lahko dosegli s tem, da bi pred drugim korakom naredili še en prevod brez parametra `-g`. Za končno datoteko bi lahko nato uporabili kodo pred in po deskriptorju metod iz tega prevoda, kjer tabela konstant ne vsebuje zapisov o lokalnih spremenljivkah. Poleg tega bi lahko iz popravljenega deskriptorja metod preprosto odstranili tabele lokalnih spremenljivk. Kot rečeno so to podatki, ki so potrebni zgolj za razhroščevanje in niso sestavni del datoteke, prevedene z običajnim prevajalnikom brez dodatnih parametrov. Lahko pa bi šli še korak dlje in bi omogočili uporabo parametrov navadnega prevajalnika tudi pri razširjenem. V tem primeru bi pred drugim korakom izvedli prevod s posredovanimi parametri navadnemu prevajalniku in ustrezno priredili izhodno datoteko.

6.2 Dodajanje zapisov v tabelo konstant

Naslednja možnost za razširitev je dodajanje zapisov v tabelo konstant. Najpreprosteje bi bilo podpreti samo dodajanje številskih ali znakovnih konstant, težava pa bi nastala pri deklaraciji novih spremenljivk. Če bi želeli, da prevajalnik v tabelo konstant doda tudi zapise spremenljivk, ki še niso bile deklarirane, bi bilo treba doreči nekaj stvari. Predvsem se tu pojavi vprašanje o vidljivosti takšnih spremenljivk. Vprašanje je, ali naj bo takšna spremenljivka vidna samo znotraj bloka, ali naj bo vidna, kot da je bila deklarirana pred `jasm` blokom. V slednjem primeru bi bilo treba dodati javansko kodo za deklaracijo spremenljivke,

da bi se lahko program prevedel brez `jasm` bloka z običajnim prevajalnikom. Nato bi morali pri optimizaciji poiskati in odstraniti odvečno kodo, ki je nastala zaradi deklaracije. Če pa bi bila spremenljivka vidna samo znotraj bloka, bi morali v tabeli lokalnih spremenljivk metode dodati njen zapis. V tem primeru bi bilo treba popraviti tako število tabel lokalnih spremenljivk kot tudi dolžino atributa `LocalVaribleTable` in še dolžino celotnega atributa `Code`.

6.3 Izdelava vtičnika

Zagotovo najbolj uporabna in tudi najbolj obsežna razširitev bi bila izdelava vtičnika za razvijalska orodja, kot sta na primer eclipse in NetBeans. Vtičnik bi moral namesto privzetega prevajalnika uporabljati izdelani razširjen prevajalnik. Za prijaznejšo uporabniško izkušnjo bi morali v sintakso dodati nov ukaz `jasm`, da ga razvijalsko orodje ne bi obravnavalo kot napako. Poleg tega bi lahko sintakso razširili še z vsemi ukazi zložne kode kot tudi njihovim številom in tipom operandov, da bi nas razvijalsko orodje sproti opozarjalo na preproste sintaktične napake brez časovno potratnega prevajanja.

Literatura

[1] Joshua Engel. Programming for the Java Virtual Machine. Reading (Massachusetts). Addison-Wesley, 1999

[2] http://andrei.gmxhome.de/bytecode/

[3] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html

[4] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[5] http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

[6] http://en.wikipedia.org/wiki/Java_virtual_machine

[7] http://www.drgarbage.com/bytecode-visualizer/

[8] http://www.eclipse.org/jdt/core/

[9] https://github.com/Andrazz/jasmCompiler